Generic Adaptive Syntax-Directed Compression for Mobile Code

Christian H. Stork
cstork@ics.uci.edu

Vivek Haldar
vhaldar@ics.uci.edu

Michael Franz
franz@uci.edu

Technical Report 00-42
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

November 2000
Revised April 2001

Abstract

We present a new approach to code compression designed as part of a larger infrastructure for mobile code deployment. Our approach operates on abstract syntax trees and can be parameterized by abstract grammars, making it applicable to any source language without further language-specific modifications. It employs a variation of prediction by partial match to drive an arithmetic coder adaptively, thereby achieving unmatched compression results without requiring prior statistical analysis. We have developed a prototype implementation and have used it to compress Java programs. It reduces the size of compressed Java classes and packages by 5–50 percent compared to the best published compression scheme, which was specifically tailored for Java. This means we reduce the size of (already compressed) Java archives by a factor of 3 to 8.

Contents

1 Introduction **2**
1.1 Vision 3

2 Compressing Abstract Syntax Trees **4**
2.1 Abstract Grammars 4
2.2 Encoding ASTs 5
2.3 Arithmetic Coding 5
2.4 Prediction by Partial Match 6
2.4.1 Adapting PPM for ASTs 7
2.4.2 Weighing Strategies 7
2.5 Compressing Constants 8

3 Implementation and Results **8**
3.1 Binary and Source Compatibility 9
3.2 Preliminary Results 10

4 Related Work **11**

5 Discussion **13**
5.1 Statistical versus Dictionary-based Encoding 13
5.2 Devising an Abstract Grammar 13

6 Conclusion **14**

1 Introduction

With the advent of mobile code, there has been a resurgent interest in code compression. Compactness is an issue when code is transferred over networks limited in bandwidth, particularly wireless ones. It is also becoming increasingly important with respect to storage requirements, especially when code needs to be stored on consumer devices. Furthermore, denser code representations can also act as an enabling technology. For example, good compression can reduce the size requirements of proof-carrying code [Nec97]. A benficial side-effect of good compression in general is that the reduction of redundancy increases the effectiveness of encryption by making statistical attacks harder. Finally, processor performance has increased exponentially over storage access time in the last decade. It is therefore reasonable to investigate compression as a means of using additional processor cycles to decrease the demand on storage access [FK97], leading to a net gain in performance.

Among the major approaches to mobile code compression are (1) schemes that use code-factoring compiler optimizations to reduce code size while leaving the code directly executable [DEM99], (2) schemes that compress object code by exploiting certain statistical properties of the underlying instruction format [EEF+97, Fra99, Luc00, Pug99], and (3) schemes that compress the abstract syntax tree (AST) of a program by using either dictionary-based [FK97] or statistical [Cam88, ECM98] approaches.

Our approach falls into the last category, or more precisely, we compress the AST of a program using novel statistical approaches. Since the AST is composed according to a given abstract grammar (AG), we are using domain knowledge about the underlying language to achieve a more compact encoding than a general-purpose compressor could achieve.

Our compression framework applies to different kinds of code. It is convenient to think of our compression algorithm as being applied to some source language, which—after decompression at the code consumer site—is compiled into native code. But generally, our scheme applies to all code formats that can be expressed in form of a grammar. Theoretically, this includes all forms of code: source code text, ASTs, intermediate representations (byte code or SafeTSA [ADFvR00] for example), and object code. Our prototype implementation compresses Java ASTs, which can then be compiled to native code, thereby circumventing compilation into byte code and execution on the JVM.

We chose to compress Java programs as a proof of concept because there already exists a sizeable body of work on the compression of Java code, especially Pugh's work on `jar`-file compression [Pug99]. This gives us a viable yardstick to gauge our results against.

Our compression scheme does not assume that source code will be re-generated at the code consumer's site. In fact, in our current implementation the decompressor interfaces directly to the GCC [Fre] backend.

In our framework, source code is required in order to generate a compressed AST and, inversely, a compressed AST possesses the intrinsic capability to regenerate the source code (deprived of comments and internal identifier names).

The prerequisite of source code availability and the benefit of ASTs being readily (re)compilable and well-optimizable, position our encoding as a good distribution format[1] for Open Source Software [OSS]. Files in our format are very compact and span different architectures, thereby reducing the maintenance effort for packaging.

1.1 Vision

Our vision for mobile code distribution and deployment is based on the notion of *code producers* and *code consumers*. The code producer distributes software as compressed ASTs, which constitute a platform-independent format at the highest possible level of abstraction. Naturally, programs distributed as ASTs are as portable as their source language allow.[2] Compression of ASTs is allowed to be computationally expensive because it is only a one-time effort performed by the code producer. Thus we can imagine augmenting the encoding with hard-to-compute but easy-to-verify annotations, e.g., alias information for further optimizations or proofs of safety properties [Nec97].

On the code consumer side, the code format has to meet several requirements: (1) short start-up time, (2) potentially pliable to more advanced optimizations, and (3) safe to execute. We meet the first requirement by providing a very dense encoding, which can be compiled directly into machine code on arrival. As shown by Franz and Kistler [FK97], the time saved for transmission (or file access) easily pays for the additional decompression and compilation effort.[3]

Since our compression format contains all the machine-readable information provided by the programmer at source language level, the runtime system at the code consumer site can readily use this information to provide optimizations and services based on source language guarantees.[4] Kistler [Kis99] uses the availability of the AST to make dynamic re-compilation at runtime feasible. Furthermore, distributing code in source language-equivalent form provides the runtime system with the choice of a platform-tailored intermediate representation. The success of Transmeta's dynamic code translation technology shows that this is a viable approach, even when starting with an unsuitable intermediate representation at a much lower abstraction level.

Lastly, high-level encoding of programs protects the code consumer against all kinds of attacks based on low-level instructions, which are hard to control

[1] Since the right to modify the source and documentation is an integral part of the Open Source philosophy, our format is no alternative to fully commented source text, but it is only meant as replacement for the binary distribution of Open Source Software. Note however that, in contrast to binary object files, our compressed ASTs still need to go through a code generation phase in order to be executable.

[2] Here we allude to portability issues caused by implicit assumptions of the source language. For example, some C programs assume an `int` to have the same size as a pointer.

[3] By now the consensus seems to be that on-the-fly compilation is preferable over bytecode interpretation. For example, in Microsoft's .NET architecture, code in intermediate language format is never interpreted but always compiled.

[4] As an example, note that the Java language provides much more restrictive control flow than Java byte code, which allows arbitrary gotos.

and verify. Our encoding also has the desirable characteristic that even after malicious manipulation it can only generate ASTs which adhere to the abstract grammar (and certain additional semantic constraints), thereby providing some degree of safety by construction. This is in contrast to byte code programs, which have to go through an expensive verification process prior to execution.

2 Compressing Abstract Syntax Trees

Computer program sources are phrases of formal languages represented as character strings. But programs proper are not really character strings, in much the sense that natural numbers are not digit strings but abstract entities. Conventional context-free grammars, i.e., *concrete grammars*, mix necessary information about the nature of programs with irrelevant information catering to human (and machine) readability. An AST is a tree representing a source program abstracting away irrelevant concrete details, e.g., which symbols are used to open/close a block of statements. Therefore it constitutes the ideal starting point for compressing a program. Note also that properties like precedence and different forms of nesting are already implicit in the AST's tree structure.

2.1 Abstract Grammars

Every AST conforms with an *abstract grammar* (AG) just as every source program conforms with a concrete grammar. AGs give a succinct description of syntactically correct programs by eliminating superfluous details of the source program.

AGs consist of *rules* (also called *productions*) defining symbols much like concrete grammars define terminals and nonterminals [Mey90]. Whereas phrases of languages defined by concrete grammars are character strings, phrases of languages defined by AGs are ASTs. Each AST node corresponds to a rule, which we will often refer to as the *kind* of node. For the purpose of a simple presentation, we will discuss only three forms of *rules*. These three forms of rules are sufficient to specify sensible AGs and are a subset of the rules used in our framework.

Two forms of rules are compound rules defining symbols corresponding to the well-known non-terminals of concrete grammars. *Aggregate rules* define AST nodes (*aggregate nodes*) with a fixed number of children. For example, the rule for the while-loop statement defines a *WhileStmt* node with two children of kind *Expression* and *Statement*:

$$\mathit{WhileStmt} \quad \triangleq \quad \mathit{Expression};\ \ \mathit{Statement}.$$

The second form of compound rules are *choice rules*, which define AST nodes (*choice nodes*) with exactly one child. The kind of child node can be chosen from a fixed number of alternatives. The following (simplified) rule says that a *Statement* node has either an *Assignment*, *IfStmt*, or *WhileStmt* child:

$$\mathit{Statement} \quad \triangleq \quad \mathit{Assignment} \mid \mathit{IfStmt} \mid \mathit{WhileStmt}.$$

The last form of rule is the *string rule*, which specifies *string nodes*. The right hand side of a string rule is the predefined STRING symbol. String rules define the equivalent of terminals in concrete grammars. String nodes contain an arbitrary string and they are the leaf nodes of the AST. To define the *Ident* node to be a string node we write:

$$\mathit{Ident} \triangleq \mathsf{STRING}.$$

User-defined symbols of AGs must be defined by exactly one rule with the exception of the predefined STRING symbol. As usual, one symbol is marked as the *start symbol* of the AG.

2.2 Encoding ASTs

In order to encode (i.e., store or transport) ASTs they need to be serialized. ASTs can be serialized by writing out well-defined traversals. We chose a depth-first traversal resulting in a pre-order representation. Such a traversal provides a linearization of the tree structure only. Several mechanisms exist in order to encode the information stored at the nodes. The most common technique pre-scans the tree for node attributes, stores them in separately maintained lists, and augment the tree representation with indices into these lists. For now, we ignore the problem of efficiently compressing strings (our only node attributes) for the sake of simplicity and assume that strings are directly encoded whenever they appear.

The actual tree representation can make effective use of the AG. Given the AG, much information in the pre-order encoding is redundant. In particular, the order and the kind of children of aggregate nodes are already known. Therefore the encoding boils down to noting the choices made at each choice node. Since the order of alternatives in choice nodes is fixed, it suffices to encode only the position $(1, 2, 3, \ldots)$ of the chosen alternative. Of course, if only one alternative is given there is "no choice" and therefore nothing needs to be encoded.

2.3 Arithmetic Coding

So far we reduced the serialization of compound rules to encoding the choices made at each choice node as an integer $c \in \{1, 2, \ldots, n\}$, where n depends on the kind of choice node and is equal to the number of given alternatives. We want to use as few bits as possible for encoding the choice c. The two options are to use Huffman coding or arithmetic coding. Using Huffman code as discussed in Stone [Sto86] is very fast, but is much less flexible than arithmetic coding. Cameron [Cam88] shows that arithmetic coding is more appropriate for good compression results and recent improved implementations [MNW98] make it also very fast.

An arithmetic coder [WNC87] is a flexible means to encode a number of choices if each alternative $i \in \{1, 2, \ldots, n\}$ has a certain probability p_i, where $\sum_{i=1}^{n} p_i = 1$ and n is given by the kind of choice node. The tuple $M =$

$(p_1, p_2, \ldots, p_n)$ is called the *model* M for the arithmetic coder. When encoding, an arithmetic coder takes a sequence of choices c_j along with their respective models M_j as argument and outputs a sequence of bits B. From this information, the arithmetic coder produces a close to optimal encoding of the sequence of choices $c_1, c_2, \ldots$. When decoding, an arithmetic coder takes the sequence of bits B and the above sequence of models $M_1, M_2, \ldots$ as arguments. For each given model M_j it then reproduces the next choice c_j. It is important to note that the model M_j can depend on all previous choices $c_1, c_2, \ldots, c_{j-1}$. The choice of models determines the quality of compression. If the probabilities are picked in an "optimal" fashion (i.e., taking "all" available information into account and adapting the probabilities appropriately) then the encoding has minimal redundancy.

A simple and fast way to chose the models is to fix the probability distributions for each kind of node. Good fixed models can be determined based on statistics over a representative set of programs.

2.4 Prediction by Partial Match

Prediction by Partial Match (PPM) [CW84] is a statistical, predictive text compression algorithm. PPM and its variations have consistently outperformed dictionary-based methods as well as other statistical methods for text compression.

Our experience shows that PPM adapts so fast to each program's peculiarities that efforts to improve compression by using statistically determined initial probabilities for the models did not yield any significant gains in compression.

PPM maintains a list of already seen string prefixes, conventionally called *contexts*. For example, after processing the string *ababc*, the contexts are the empty context, *a, b, c, ab, ba, bc, aba, bab, abc, abab, babc,* and *ababc*. For each context PPM maintains a list of characters that appeared after the context. PPM also keeps track of how often the subsequent characters appeared. So in the given example the counts of subsequent characters for, say, *ab* are *a* and *c* both with a count of one. Normally, efficient implementations of PPM maintain contexts dynamically in a *context trie* [CT97]. A context trie is a tree with characters as nodes and where any path from the root to a node represents the context formed by concatenating the characters along this path. The root node does not contain any character and represents the empty context (i.e., no prefix). In a context trie, children of a node constitute all characters that have been seen after its context. In order to keep track of the number of times that a certain character followed a given context, the number of its occurrences is noted along each edge. Based on this information PPM can assign probabilities to potential subsequent characters.

The length of contexts is also called their *order*. Note that contexts of different order might yield different counts leading to varying predictions. Different strategies have been devised to blend the information given by contexts of different orders.

2.4.1 Adapting PPM for ASTs

When applying PPM to trees the first problem to solve is the definition of contexts for ASTs. We chose a simple definition:

> The *context* of an AST node is defined as the concatenation of its ancestors on the path to the root.

One consequence of this definition is that the order of contexts is bounded by the depth of the AST. Our alphabet corresponds to the rules, i.e., symbols, of the AG because our modified PPM algorithm treats AST nodes like the original PPM algorithm treats characters.[5] The PPM algorithm is applied to the nodes as they appear while traversing the AST in depth-first order.

PPM maintains a set of nodes in the context trie called *active nodes*. Active nodes mark the positions, where the current contexts end. The root of the trie, representing the empty context, is always active. When the AST traversal descents to a child node, new nodes in the context trie are created as children of active nodes. This corresponds to the familiar addition to the current contexts. However, whenever the AST traversal proceeds from a leaf node to an internal node (as in DFS) suffixes of the current contexts are annihilated, i.e. the input seen by the modified PPM does not consist of contiguous characters anymore. This changed requirement makes it necessary to partly *pop* contexts, i.e., all nodes marked as active (except the root) in the context trie are moved up one node to their parents. (The root always stays active.) This ensures that all children of a node N in the AST appear as children of N in the context trie too. This works because we traverse the AST in depth-first order while building up contexts.

We adapted the unbounded variant of the PPM algorithm (PPM*) [CT97] for our implementation. Given our definition of context together with the way we pop contexts, the depth of our context trie is bounded by the AST's depth. Therefore we don't have to worry about unlimited growth of the context trie in spite of using PPM*.

2.4.2 Weighing Strategies

In order to generate the model for the next encoding/decoding step, we look up the counts of symbols seen after the current context in the context trie. Since the active nodes, to which we have direct pointers, correspond to the last seen symbol, this is a fast lookup and does not involve traversing the trie. These counts can be used in several ways to build the model. Normally, the context trie contains counts for contexts of various orders. We have to decide how to weigh these predictions of various orders to get a suitable model. The trade-off is that shorter contexts occur more often, but fail to capture the specificity and sureness of longer contexts (if the same symbol occurs many times after a very

[5] Note that if an aggregate node has several children of the same kind then their position is relevant for the context. Since this does not happen often, we have not implemented this refinement yet.

long context, then the chance of it occurring again after that same long context is very high), and longer contexts do not occur often enough for all symbols to give good predictions. Note that the characteristics of AST contexts differ from text contexts—AST contexts are bound by the depth of the AST and tend towards more repetitions since the prefixes of nodes for a given subtree are always the same

We tried various weighing strategies, and our experiments indicate that ignoring predictions made by order 0 contexts (which are simply relative frequencies of symbols, and form the first level of the context trie) and weighing all other predictions equally yields the best compression. This will be explored further in an upcoming paper.

Note that this approach for adapting PPM to compress abstract syntax trees is general enough to compress any kind of tree, and not just ASTs.

2.5 Compressing Constants

A sizable part of an average program consists of constants like integers, floating-point numbers, and, most of all, string constants. String constants in this sense encompass not only the usual string literals like "`Hello World!`" but also type names (e.g., `java.lang.Object`), field names and more. In our simplified definition of AGs, we used the predefined STRING symbol to represent constants within ASTs. However, when observing the use of strings in ASTs of typical programs, it is apparent that many strings are used multiple times. Therefore it saves space to encode the different strings once and refer to them at later occurrences. Such a reference is an index into a list of strings. The higher the number of strings is, the more bits are needed to encode the corresponding index. By distinguishing different kinds of strings (e.g., type names, field names, and method names) different lists of strings can be created. These split lists are each smaller than a global list. Given that the context determines which list to access, references to strings in split lists require less space to encode. As these considerations show, context-sensitive (as opposed to context-free) information such as symbol tables can be encoded and compressed at varying degrees of sophistication.[6] Our framework provides the facility of so-called *pools*, which embody different ways of compressing, maintaining, and accessing lists of constants.

3 Implementation and Results

Our current implementation is a prototype written in Python [Pyt] consisting of roughly 40 modules handling AGs, ASTs, and their compression/decompression. In order to compare our compression results to other established methods we chose to compress Java programs. Our Java frontend is written in Java and uses the Barat framework [BS98] for parsing. We devised an AG for Java, which is

[6]Note that conventional symbol tables can conveniently be expressed as some kind of AST with the appropriate string nodes.

both suitable for easy generation from Barat's internal representation of Java programs and suitable to generate a dense encoding. A visitor for walking Barat's AST was then adapted to output a Lisp-like textual representation of the AST according to our AG. The textual representation of the AST is then parsed and compressed by our Python prototype. This compressed binary file can be stored or sent over networks. After decompressing the binary file, the prototype can interface directly to any kind of backend. Currently, we work on the integration with GCC as code-generating backend.

It is natural to implement most of our AST processing with the *visitor* design pattern [GHJV95]. Visitors are used to walk the AST and perform different tasks on the tree, e.g., gathering all occurring constants or computing the probabilities for the arithmetic coder. Visitors are a good means to separate and recombine different passes over the AST. We evolved the visitor design pattern into the *weaver/yarn* pattern, which allows us to re-use the same visitor code for compressing and decompressing despite the fact that the AST is being built by the code consumer while being traversed by several visitors (i.e., yarns) in lockstep. This design pattern will be described in a separate paper. This architecture has helped us tremendously during the development of the prototype.

In our implementation we provide generic ways to mark and reference nodes within the AST. This gives us the means to allow very concise augmentations of the AG that specify how to encode constructs like labels or local (i.e., statically-scoped) variables very effectively. Furthermore, we provide generic building blocks (*pools*) to handle string, integer, and floating point constants.

All information necessary to specify the AST's compression and decompression is condensed into one configuration file. The configuration file contains the AG augmented with additional information, e.g., on how to compress different kinds of constants. Given the availability of our framework at the code producer and consumer sites, the only requirement for supporting the compression/decompression of an additional language is that identical copies of the configuration file are present at both sites.[7]

3.1 Binary and Source Compatibility

The Java Language Specification (JLS) [GJSB00] devotes the entirety of chapter 13 on *binary compatibility* of Java class files. Binary compatibilty ensures that class files (i.e., binaries), which have been compiled against other class files, will still link correctly with newer versions of the accessed class files. This enables library vendors to update their libraries without forcing client code to be recompiled. In order to achieve this goal the vendor must restrict the library class changes to the list of *binary compatible changes* defined in the JLS.

The AG currently used in our framework fullfills most requirements for binary compatibility.[8] It is based on Barat's representation of Java classes and in-

[7]Note that in order for the code consumer to deploy the transported code, it still needs to compile it into some executable format.

[8]We do not yet replace fields that are final and initialized at compile-time with their constant value and we do not yet resolve methods/constructors at compile-time to their qualifying

Class Name	Size in Bytes					CAST/Pugh
	Class File	Gzip	Bzip2	Pugh	CAST	
`ErrorMessage`	305	256	270	209	105	50%
`CompilerMember`	1192	637	641	396	230	58%
`BatchParser`	4939	2037	2130	1226	1069	87%
`Main`	11363	5482	5607	3452	3295	95%
`SourceMember`	13809	5805	5705	3601	2988	83%
`SourceClass`	32157	13663	13157	8863	7849	89%

Table 1: File size comparision of compressed AST files (CAST) with class files from `sun.tools.javac` compressed using alternative techniques.

Package Name	Size in Bytes					CAST/Pugh
	Jar	Gzip	Bzip2	Pugh	CAST	
`sun.tools.javac`	36787	32615	30403	18021	14070	78%
`jess`	232041	133146	97852	48331	31083	64%

Table 2: File size comparision of compressed collections of classes from two Java packages.

terfaces, which among other advantages removes ambiguities like the ones caused by the type-import-on-demand declaration (e.g., `import some.package.*;`) by performing a static name analysis and always referring to fully-qualified type names.

3.2 Preliminary Results

In this section we compare the compression results of our prototype against other general-purpose and special-purpose compression algorithms. We split the comparison in two parts: First we measure compression of single classes and, second, we measure compression of collections of classes as they appear in Java packages or `jar`-files. Our basic Java AG defines how to represent Java classes as ASTs. With a two-line addition, our original AG can also deal with collections of classes as present in packages or `jar`-files. These collections of classes share the same pools (lists of strings, etc.) thereby reducing redundancy caused by entries, which appear in several classes. We can use our framework with the extended AG to compresss the classes contained in `jar`-files. This gives us the basis for a good comparison with Pugh's work.

The Java code chosen for compression is the Java compiler package from Sun (Linux Blackdown Version 1.1.2) and Jess, a rule engine and scripting environment ([Jes], version 5.1). Both packages were used in the SPEC JVM98 Benchmark suite [Sta] and they are the only ones thereof for which the source

type of invocation plus their signature.

code is available. We use the most current (source) versions of these packages as indicated above. In case of Jess, we compressed all classes that are part of the distribrution, i.e., including the subpackage and example classes. The class files were compiled under javac (Linux Blackdown Version 1.1.2) with all debugging information excluded (`-g:none` option). They were not stripped with a tool equivalent to Pugh's `StripZip` program since we want to give a comparison with what is in common use today. We compare only the compression of Java classes proper by eliminating all other resource files including the manifest.

We choose primarily Pugh's compression scheme [Pug99] for comparison because, to our knowledge, it provides the best compression ratio for Java archives (and class files) and it is freely available for educational purposes. It should be noted that Pugh actually designed his compression scheme for `jar` files, which are collections of (mostly) class files. His algorithm therefore does not perform as well on small files as it does on bigger ones. We fed the evaluation version 0.8.0 of Pugh's Java Packing tool with `jar`-files generated with the `-M` option (no manifest).[9] The other comparable compression scheme is syntax-oriented coding [ECM98]. But for this scheme there are no detailed compression numbers available, only an indication that the average compression ration between their format and regular class files is $1 : 6.5$.

We furthermore compare our results with two widely available general purpose compression algorithms, gzip and bzip2. Collections of classes (Table 2) have been `tar`'ed before applying `gzip` or `bzip2`.

The comparision of compressing Java classes is presented in Table 1 and the comparision for collections of classes is presented in Table 2. Our choice of single classes tries to be representative of the sizes of classes in the SPEC JVM98 Benchmark suite. The resulting numbers show that our compression scheme is an improvement by 5–50% over Pugh's results, which translates to a compression of regular `jar`-files by a factor of 3 to 8, roughly. The results indicate that we compress very well for either very small classes or larger collections of classes. Some more statisitcal investigation is needed to precisely analyse and, ultimately, enhance our compression results.

4 Related Work

The initial research on syntax-directed compression was conducted in the 1980's primarily in order to reduce the storage requirements for source text files. Contla [Con81, Con85] describes a coding technique essentially equivalent to the technique described in section 2.2. This reduces the size of Pascal source to at least 44% of its original size. Katajainen et. al. [KPT86] achieve similar results with automatically generated encoders and decoders. Al-Hussaini [AH83] implemented another compression system based on probabilistic grammars and LR parsing. Cameron [Cam88] introduces a combination of arithmetic coding with the encoding scheme from section 2.2. He assigns fixed probabilities to

[9] This means in case of compressing classes we first make a `jar`-file from an individual class file and then compress the resultant `jar`-file using Pugh's tool.

alternatives appearing in the grammar and uses these probabilities to arithmetically encode the pre-order representation of ASTs. Furthermore, he uses different pools of strings to encode symbol tables for variable, function, procedure, and type names. Deploying all these (even non-context-free) techniques he achieves a compression of Pascal sources (including comments) to 10–17% of their original size. Katajainen and Mäkinen [KM90] present a general survey of tree compression mentioning the above methods. It seems that before this paper all of the above four efforts were pursued independently of each other. Tarhio [Tar95] suggests the application of PPM to drive the arithmetic coder in a fashion similar to ours. He reports increases in compression of Pascal ASTs (excluding constants, i.e., pools of strings, etc.) by 20% compared to a technique close to Cameron's.[10] Cheney [Che00] suggests applying PPM in the context of term compression.

All of these techniques are concerned only with compressing and preserving the source text of a program in a compact form and do not attempt to represent the program's semantic content in a way that is well-suited for further processing such as dynamic code generation or interpretation ([KPT86] even reflects incorrect semantics in their tree). Franz [Fra94, FK97] was the first to use a tree encoding for (executable) mobile code. He uses a dictionary-based encoding to compress the abstract syntax tree of Oberon programs.

Even though seemingly placed in the same application domain, research on "code compression" [EEF+97, Fra99, Luc00, DEM99] is generally not comparable to the above line of work on source text and AST compression. The reason is that code compression focuses much more on the specifics of machine code like choice of op codes, operand formats, lack of apparent high-level structure, and so on. Nevertheless, will we try to identify potential overlap between our work and other work on code compression.

Java, currently the most prominent mobile code platform, attracted much attention with respect to compression. Horspool and Corless [HC98] compress Java class files to roughly 36% of their original size using a compression scheme specifically tailored towards Java class files. In a follow-up paper Bradley, Horspool, and Vitek [BHV98] further improve the compression ratio of their scheme and extend its applicability to Java archives (`jar`-files). A better compression scheme for `jar`-files was proposed by Pugh [Pug99]. His format is typically 1/2 to 1/5 of the size of the corresponding compressed `jar`-file (1/4 to 1/10 the size of the original class files). Pugh offers his tool for free evaluation.

All of the above Java compression schemes start out with the byte code of Java class files, in contrast to the source program written in the Java programming language. Eck, Changsong, and Matzner [ECM98] employ a compression scheme similar to Cameron's and apply it to Java sources. They report compression down to around 15% of the original source file, although more detailed information is needed to assess their approach.

[10] Unfortunately, we learned of Cameron's and Tarhio's work only after we developed our solution independently of both.

5 Discussion

This section discusses additional issues related to our compression scheme.

5.1 Statistical versus Dictionary-based Encoding

The only other AST compression scheme for mobile code [FK97] uses a dictionary-based encoding. Our statistical encoding scheme diverts from this approach because the compression ratio of dictionary-based compression seems lower and when we tried to guarantee additional semantic constraints while decoding, the cost of maintaining valid entries in a dictionary became unbearable in terms of time and complexity.

In general, dictionary-based compression has the disadvantage that, in order for compression to succeed, an exact match from the dictionary needs to be found. Therefore either the dictionary needs to be very big to provide many potential matches or the matching algorithm needs to be rather complicated to allow some kind of "fuzzy" matching (possibly mimicking some statistical approach).

5.2 Devising an Abstract Grammar

Essentially, the combination of arithmetic coding and PPM gives full freedom in the choice of grammar. Stone [Sto86] and Cameron [Cam88] propose different grammar expansion techniques in order to make nested choices accessible to statistical approaches. But since PPM keeps track of the nesting (i.e., "expansion") of rules in its context we do not have to worry about rewriting the grammar. More specifically, with respect to Al-Hussaini's scheme Stone [Sto86] discourages layering of rules in AGs for two reasons: (1) When using Huffman encoding the "quality" of the encoding can only become worse, and (2) layered rules used in different contexts waste potential for a better model. With our compression scheme, we alleviate the first problem by using an arithmetic coder and the second by proper use of PPM.

Another advantage is that due to arithmetic coding we do not need to chose selection rules with 2^n choices in order to encode the choices efficiently within n bits. Put another way: "Layering" of choice rules can not hurt us as feared by Stone who assumed Huffman coding as the best encoding method.

The freedom to choose an arbitrary abstract grammar can be used to tailor the grammar towards other desirable properties:

- Allowing easy generation of the AST from a given frontend.
- Facilitating fast generation of good code.
- Supporting annotations such as proofs for proof-carrying code.

6 Conclusion

Our results indicate that our generic approach to syntax-directed AST compression is not only feasible but actually outperforms existing methods in compression effectiveness. We compared our compression scheme to Pugh's Java-specific compression scheme, which is the best published so far for Java, and improved compression by 5-50%. Our main contribution is to show that compressing abstract syntax trees outperformes other approaches to high-level code compression in terms of code density, even though it is the more generic approach.

Currently our research is focused on improving the compression ratio by exploring extended context definitions and different blending schemes for PPM. Additionally, we aim at enhancing the genericity of our framework and then we will focus on improving the speed of compression and decompression.

Acknowledgments The authors would like to thank Peter Fröhlich for his many contributions to this paper, Peter Housel, Niall Dalton, and Naomi Carpenter for their reviews, and Dan Hirschberg for interesting discussions. We want to express our gratitude towards Sumit Mohanty, Bratan Kostov, and Ziemowit Laski, who contributed to this project in its early stages, and Sergiy Zhenochin, who is currently working on integrating GCC as our backend.

This effort is partially supported by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0536 and by the National Science Foundation, Program in Operating Systems and Compilers, under grant CCR-9901689.

We dedicate this work to Bratan Kostov.

References

[ADFvR00] Wolfram Amme, Niall Dalton, Michael Franz, and Jeffery von Ronne. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. Technical Report 00-43, University of California, Irvine, November 2000.

[AH83] A. M. M. Al-Hussaini. *File compression using probabilistic grammars and LR parsing.* PhD thesis, Loughborough University, 1983.

[BHV98] Quetzal Bradley, R. Nigel Horspool, and Jan Vitek. JAZZ: An efficient compressed format for Java archive files. In *Proceedings of CASCON'98*, pages 294–302, Toronto, Ontario, November 1998.

[BS98] Boris Bokowski and André Spiegel. Barat – A front-end for Java. Technical Report B-98-09, Freie Universität Berlin, December 1998.

[Cam88] Robert D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, July 1988.

[Che00] James Cheney. Statistical models for term compression. In *Data Compression Conference*, page 550, 2000.

[Con81] Jose Felipe Contla. *Compact Coding Method for Syntax-Tables and Source Programs*. PhD thesis, Reading University, England, 1981.

[Con85] J. F. Contla. Compact coding of syntactically correct source programs. *Software-Practice and Experience*, 15(7):625–636, 1985.

[CT97] John G. Cleary and W. J. Teahan. Unbounded length contexts for PPM. *Computer Journal*, 40(2/3):67–75, 1997.

[CW84] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[DEM99] Saumya Debray, William Evans, and Robert Muth. Compiler techniques for code compression. In *Workshop on Compiler Support for System Software*, May 1999.

[ECM98] Peter Eck, Xie Changsong, and Rolf Matzner. A new compression scheme for syntactically structured messages (programs) and its applications to Java and the Internet. In *Data Compression Conference*, page 542, 1998.

[EEF+97] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code compression. In *Proceedings of the ACM Sigplan '97 Conference on Programming Language Design and Implementation*, pages 358–365, 1997. Published as Sigplan Notices, 32:5.

[FK97] M. Franz and T. Kistler. Slim Binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[Fra94] M. Franz. *Code-Generation On-the-Fly: A Key to Portable Software*. PhD thesis, ETH Zurich, March 1994.

[Fra99] C. W. Fraser. Automatic inference of models for statistical code compression. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1999.

[Fre] Free Software Foundation. GNU Compiler Collection. See online at http://gcc.gnu.org/ for more information.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

[HC98] R. Nigel Horspool and Jason Corless. Tailored compression of Java class files. *Software–Practice and Experience*, 28(12):1253–1268, October 1998.

[Jes] Jess, the Java Expert System Shell. See online at http://herzberg.ca.sandia.gov/jess/ for more information.

[Kis99] Thomas Kistler. *Continuous Program Optimization*. PhD thesis, University of California, Irvine, November 1999.

[KM90] Jyrki Katajainen and Erkki Mäkinen. Tree compression and optimization with applications. *International Journal of Foundations of Computer Science*, 4(1):425–447, 1990.

[KPT86] J. Katajainen, M. Penttonen, and J. Teuhola. Syntax-directed compression of program files. *Software-Practice and Experience*, 16(3):269–276, 1986.

[Luc00] S. Lucco. Split stream dictionary program compression. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2000.

[Mey90] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. PHI Series in Computer Science. Prentice Hall, 1990.

[MNW98] Alistair Moffat, Radford M. Neal, and Ian H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, 1998.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[OSS] Open Source Software. See online at http://www.opensource.org for more information.

[Pug99] William Pugh. Compressing java classfiles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 247–258, 1999.

[Pyt] Python programming language. See online at http://www.python.org for more information.

[Sta] Standard Performance Evaluation Corportation. SPEC JVM98 benchmarks. See online at http://www.spec.org/osg/jvm98 for more information.

[Sto86] R. G. Stone. On the choice of grammar and parser for the compact analytical encoding of programs. *Computer Journal*, 29(4):307–314, 1986.

[Tar95] Jorma Tarhio. Context coding of parse trees. In *Data Compression Conference*, page 442, 1995.

[WNC87] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.